Game Physics – Programming Exercise

Exercise 3 – *Collision Detection*

Task Overview

In this exercise you should implement a **SphereSystemSimulator** and an interactive "balls in a box" scene. The **SphereSystemSimulator** should handle the collisions between pairs of balls in the box, first with brute force collision detection, then with uniform grid accelerated collision detection, and optionally, a KD-tree accelerated one. The scene, acceleration data structures and an equation for repulsive force is given in the Table 3.1.

Implement the Simulator in a way that you can run multiple "balls in a box" scenes at the same time. In your simulator class, you can have multiple member objects from class SphereSystem (which you should implement). Thus, you should be able to visually verify that every method leads to the same result (e.g. by displaying each of them in a different color). Also, for one of the demonstration tasks below, you'll need to run your simulations *without* a visual display.

Details of requirements (Demo requirements, Deliverables) can be found below.

Table 3.1, "balls in a box" scene

Simulate n balls (solid spheres) with identical mass m and radius r which are contained in a bounding box geometry. Each ball is simply defined through a position and a velocity.

If two balls collide, i.e. the distance $d = \| p_2 - p_1 \|$ between the two ball centers p_1 and p_2 is less than $2r$, a penalty repulsion force will be applied to both balls.

The strength of the force is given by the kernel ($\lambda > 0$ is an appropriate scaling factor):

$$f(d) = \begin{cases} \lambda(1 - \frac{d}{2r}), & \text{if } d < 2r \\ 0, & \text{else} \end{cases}$$

Acceleration data structures for collision detection:

- **Naïve,** $O(n^2)$: Test all balls against all other balls. Use this as a reference, all accelerated methods have to yield the same result (up to float precision)!
- **Uniform grid,** $O(n)$: Use a uniform grid with spacing $h = 2r$ that covers your whole domain. In each time step, sort the balls into the grid such that each cell contains a list of all balls whose center lies inside the respective cell (each ball gets stored in exactly one cell). Then, in order to search for balls colliding with a given ball A, you only have to search in the cell of ball A and its surrounding neighbor cells. Note that this is a fast but specialized technique.
- **Optional, KD-tree:** $\approx O(n \log n)$: Use a KD-tree that is (re-)built in every time step from the current ball distribution. Each leaf of the tree should contain a list of all balls whose bounding

box overlaps the leaf (each ball can be stored multiple times in different leafs). Note that the KD-tree does not have to be perfectly balanced (i.e., the depth of the leaf nodes can vary).

Recommendations & Tips:

1. **Balls Simulation.** The basic simulation is very similar to the mass-spring systems from Exercise 1. In principle, you can start with your existing code and replace your MassSpringSystem with your new SphereSystem, and replace the spring forces with the repulsion forces introduced above.
2. **Support multiple objects.** Make sure that you can run two or more independent sets of balls at the same time, each with their own the positions and velocities. You probably need multiple SphereSystem objects in your simulator class.
3. **Stability.** This system of balls is prone to become instable, especially if several balls stack. Therefore, you should use a second order time integrator (Leap Frog or Midpoint Rule) in combination with small time steps and damping forces (like for mass spring systems).
4. **Kernel**. The linear force kernel given above is a very simple choice. If you want, you can play around with other kernels. E.g., try making the repulsion force proportional to $1/d^2$ (imitating electrostatic forces). We offer 5 kernels in the "SphereSystemSimulator.cpp" file.
5. **Uniform grid.** For best performance, and instead of creating a real list object for every grid cell, you can pre-allocate a fixed number of slots for every grid cell (e.g., 10 slots for a maximum of 10 balls per cell). This can be done in a single array with $10 \cdot m$ elements (m being the number of cells in the grid), using a second array of size m with counter variables to keep track of how many slots are occupied in every cell.
   Note (optional): In order to get actual $O(n)$ behavior, you cannot "touch" every grid cell in each time step (or else you have $O(n+m)$, which can be quite slow for $m \gg n$). Depending on your implementation, you may have to avoid this explicitly, e.g., by building a list of occupied grid cells (cells which contain at least one ball) when sorting the balls into the grid.
6. **KD-tree**. While it isn't the fastest choice, the simplest way to implement the KD-tree is in an object-oriented fashion, e.g. two derived classes for leafs and inner nodes which inherit from the same base class.
   To store lists of balls in the leafs, you can use std::vector<T> with e.g. T=int to store the ball indices, or with T=MyBallType* to store pointers to your ball objects (std::vector is an array list, see member function push_back()).
   To split two branches, you can use the vector4Dim data structure (in file vector4d.h, please download the new version from moodle page to replace the old one in the template project) to represent the splitting plane, where x, y and z stand for the normal of the plane, and the t stands for the intercept.
   When looking for collisions, you can keep track of processed pairs of balls by storing them in a set of pairs: std::set<std::pair<T,T>> (same choices for T). Note, however, that the set class will treat (i,j) and (j,i) as two different pairs, so you have to make the pairs unique by, e.g., sorting them into ascending order.
7. **Timestep size & performance.** For measuring the performance of your simulation, use the timer function provided with the exercise on the moodle page. *Caution:* Rendering a huge amount of spheres with DirectXTK can be very slow, so running the simulations without a display for performance comparisons is crucial!

Demo requirements:

Your submission should contain the following demos and should support interactive switching between these demos:

- **Demo 1, simulation with naïve collision detection:**
  - Download the Simulations.zip file from moodle page and update files in your project. *Caution:* we updated the main.cpp to add new headers and defines for the SphereSystemSimulator.
  - Implement your SphereSystemSimulator, you can start from your mass-spring systems.
  - Build the scene according to Table 3.1. Parameters (mass m, number n and radius r) should be changeable through user interaction.
  - Add gravity as external force first. After that, try more external forces, for e.g., add a force according to the mouse motion.
  - Use Naïve method to detect collisions between every two balls and the collision between balls and the bounding box. Add collision force onto every ball according to the give force kernel.
  - Simulate the "balls in a box" scene with a second order time integrator (Leap Frog or Midpoint Rule) in combination with small time steps and damping forces (like for mass spring systems).

- **Demo 2, simulation with accelerated collision detection:**
  - According to Table 3.1, build your Grid data structure and use it to accelerate your collision detection **(sample video on moodle page)**.
  - Optionally, according to Table 3.1, build your KD-tree data structure and use it to accelerate your collision detection.
  - Demonstrate your scene with user interaction.

- **Demo 3, performance comparison:**
  - **Accuracy comparison.** Initialize a simulation with n=100 balls. From the same setup, run the simulation with Naïve and accelerated collision detection methods at the same time. You can use two (or three, if KD-tree is implemented) objects of the SphereSystem class. For every time step, your simulator updates the two objects independently. You can add a checkbox to enable or disable the rendering of the 100 balls from every SphereSystem. Every SphereSystem should has its own color. Make sure that visually there is no difference for the first ca. 100 time steps. (Round off errors can accumulate over time, typically ca. 1e-05 for each step. So when running the simulation for a very long time, you might see slightly different end results. **There is a sample video on moodle page**.)
  - **Speed comparison.** Compare the performance of the three methods without displaying the result in the viewport (e.g. by running the simulation for a fixed number of steps by the press a start button, and printing the result to the command line). Measure the performance for every method for n=100, n=1000, and n=10000. Create a table with these numbers in a text file called "performance.txt", and submit this file in the root directory of your implementation. Demonstrate that your accelerated method outperforms the **naïve** approach. For larger values of *n*, your simulation might not run at interactive frame rates anymore. (For the KD-tree, a large *n* is typically necessary to demonstrate a speed-up.)

- In this demo, you can disable the parameter updates from the user interface (such as the mass, radius, lambda, damping and so on).

Deliverables

The deadline for this exercise is on **Dec. 22, 23:59**.

If you are using Bitbucket/GitHub, please send your repository link to your tutor, and make sure that our tutor has the permission accessing it. If you already shared the repository with your tutor, a notification email before the deadline is still necessary, pointing out which commit the tutor should use as your final version.

If you are not using Bitbucket/GitHub, please pack all your source files (.h and .cpp) and project files (.vcxproj) under the "Simulations" directory into a zip file, and name it "Group??_Ex3_VS201?.zip", and sent it to your tutor. Make sure not to include the compiler temporary files (they will be under the Simulations/Win32/ directory). Your package should be smaller than 100kb.